5 - Classes et objets

1. Utilité de la programmation par objets
2. Syntaxe de la création et de l'utilisation de classes et instances
3. Classes paramétrées
4. Classes et définitions récursives
5. Héritage de classes

5.1 - Utilité de la programmation par objets

- On peut créer de nouveaux objets Python comme des **instances de classes** qui sont définies par les **attributs** et les **méthodes** s'appliquant à ces objets.

- Les **attributs** d'une classe sont des propriétés permettant de décrire ses instances.

- Les **méthodes** sont des fonctions qu'on peut appliquer aux instances de la classe.

5.1 - Utilité de la programmation par objets

- L'utilisation de classes permet de **personnaliser le typage**, une classe pouvant être vue comme un type personnalisé et une instance de celle-ci comme un objet typé.

- Le mécanisme de l'**héritage** permet de définir de nouvelles classes à partir d'anciennes classes en les particularisant.

5.2 - Syntaxe de la création et de l'utilisation de classes et instances

- La syntaxe pour créer une nouvelle classe est la suivante :

  class < *nom de classe* > ***:***

  < *commentaire de description* >

  < *bloc d'instructions* >

  La partie < *commentaire de description* > est une chaîne de caractères qui est une description de la classe. La partie < *bloc d'instructions* > comprend en particulier les initialisations d'attributs et les définitions de méthodes.

5.2 - Syntaxe de la création et de l'utilisation de classes et instances

- La syntaxe de la définition d'une méthode est celle de la définition d'une fonction avec un premier paramètre obligatoire *self* qui représente l'instance de la classe concernée par l'appel de la méthode.

- L'utilisation d'un attribut *attr* d'un objet *obj* s'effectue à l'aide de l'expression *obj.attr*.

- L'utilisation d'une méthode *m* associée à un objet *obj* s'effectue à l'aide de l'appel de fonction $obj.m(arg_1, arg_2, \ldots, arg_n)$.

5.2 - Syntaxe de la création et de l'utilisation de classes et instances

```
>>> class Rectangle :
    u"Rectangles géométriques"
   #attributs
    longueur = None
    largeur = None
    #méthodes
    def perimetre(self) :
        return (self.longueur +self.largeur)*2
    def aire(self):
        return self.longueur *self.largeur
```

```
>>> r1 = Rectangle()
>>> r1.longueur = 5
>>> r1.largeur = 3
>>> print  u"aire : ", r1.aire(),   u"  périmètre : " , r1.perimetre()
aire :15    périmètre : 16
```

5.3 - Classes paramétrées

- Il peut être utile de paramétrer la création d'une instance d'une classe. Cela permet d'initialiser les valeurs des attributs d'une instance au moment de sa création.

- Dans la définition de la classe correspondante, il faut définir une méthode spéciale, qui doit avoir le nom *__init__* et qui est exécutée à la création de chaque instance de la classe.

- La syntaxe de la définition de *__init__* est la suivante :

  def __init__(*self, arg_1, arg_2, …, arg_n* ***):***

  < bloc d'instructions >

  Dans cette définition, le premier paramètre *self* représente l'instance de la classe en cours de création.

- L'instruction de création d'une instance *obj* d'une classe *C* a alors la forme : *obj = C(arg_1, arg_2, …, arg_n)*

5.3 - Classes paramétrées

```
>>> class Rectangle :
    u"Rectangles géométriques"
    def __init__(self, lo, la)
      self.longueur = lo
      self.largeur = la
    def perimetre(self) :
       return (self.longueur +self.largeur)*2
    def aire(self):
       return self.longueur *self.largeur
```

```
>>> r1 = Rectangle(5,3)
>>> print  u"aire : ", r1.aire(),   u"  périmètre : " , r1.perimetre()
aire :15    périmètre : 16
>>> r1.longueur = 10
>>> print  u"aire : ", r1.aire(),   u"  périmètre : " , r1.perimetre()
aire :30    périmètre : 26
```

5.4 - Classes et définitions récursives

- Il est parfois nécessaire de définir les instances d'une classe à l'aide d'attributs dont la valeur contient des instances de la même classe.

- Dans ce cas, la définition des méthodes associées est souvent **récursive** : la définition d'une méthode contient un appel à cette même méthode.

5.4 - Classes et définitions récursives

```
>>> class Liste :
    def __init__(self, t= "", q = None):
        self. tete = t
        self.queue = q
    def longueur(self):
        if self.queue == None :
            return 1
        else:
            return self.queue.longueur() +1
    def afficher(self):
        if self.queue == None :
            return [self.tete]
        else :
            return [self.tete] +self.queue.afficher()
```

```
>>> l1 = Liste("Marie")
>>> l2 = Liste("aime", l1)
>>>l3 = Liste("Jean",l2)
>>> print l3.longueur()
3
>>> print l3.afficher()
['Jean', 'aime', 'Marie']
```

5.5 - Héritage de classes

- Une classe peut être définie par **héritage** d'une autre classe. Elle reprend alors les attributs et les méthodes de celles-ci mais elle peut en ajouter pour particulariser la classe initiale.

- La syntaxe de l'entête de définition d'une classe C_1 qui hérite d'une classe C_2 est la suivante : ***class*** $C_1(C_2)$ ***:***

- Dans la définition d'une classe qui en hérite d'une autre, on peut redéfinir des méthodes de celle-ci. On parle alors de **surcharge** de méthode.

5.5 - Héritage de classes

```
>>> class MotEtiquete :
    def __init__(self, m = "", e = "") :
        self.mot = m
        self.etiquette = e
    def affichage(self) :
        return (self.mot, self.etiquette)

>>> class ListeMotsEtiquetes(Liste) :
    def __init__(self, me=MotEtiquete(), q=None) :
        Liste.__init__(self, me,q)
```

```
>>> me1 = MotEtiquete("Jean", "NP")
>>> me2 = MotEtiquete("aime", "V")
>>> me3 = MotEtiquete("Marie", "NP")
>>> lme1 = ListeMotsEtiquetes(me3)
>>> lme2 = ListeMotsEtiquetes(me2, lme1)
>>> lme3 = ListeMotsEtiquetes(me1, lme2)
>>> print lme3.afficher()
[<__main__.MotEtiquete instance at 0xcd7300>, <__main__.MotEtiquete instance at
    0xcd7328>, <__main__.MotEtiquete instance at 0xcd7350>]
```

5.5 - Héritage de classes

```
>>> class ListeMotsEtiquetes(Liste):
    def __init__(self, me=MotEtiquete(), q=None):
        Liste.__init__(self, me,q)
    def afficher(self):
        if self.queue == None :
            return [self.tete.affichage()]
        else :
            x = [self.tete.affichage()]
            x.extend(self.queue.afficher())
            return x
```

```
>>> me1 = MotEtiquete("Jean", "NP")
>>> me2 = MotEtiquete("aime", "V")
>>> me3 = MotEtiquete("Marie", "NP")
>>> lme1 = ListeMotsEtiquetes(me3)
>>> lme2 = ListeMotsEtiquetes(me2, lme1)
>>> lme3 = ListeMotsEtiquetes(me1, lme2)
>>> print lme3.afficher()
[('Jean', 'NP'), ('aime', 'V'), ('Marie', 'NP')]
```

5.6 - Exercices

1. On veut écrire une classe *MotFlechi* avec 3 attributs : *mot, lemme, categorie*. Le premier correspond au mot fléchi, le second à son lemme et le dernier sa catégorie grammaticale. On a une seule méthode *afficher* qui permet d'afficher les informations sous forme d'un triplet.
   Dériver de la classe *MotFlechi* les classes *VerbeFlechi, NomFlechi* et *AdjectifFlechi* qui prennent en compte les paramètres de flexion spécifiques aux verbes, noms et adjectifs.

2. On veut écrire une classe pour les arbres syntaxiques. Un arbre syntaxique est un arbre étiqueté par une catégorie grammaticale, si ce n'est pas une feuille, ou par un mot, si c'est une feuille. On se restreint aux arbres où les nœuds ont au plus deux fils.

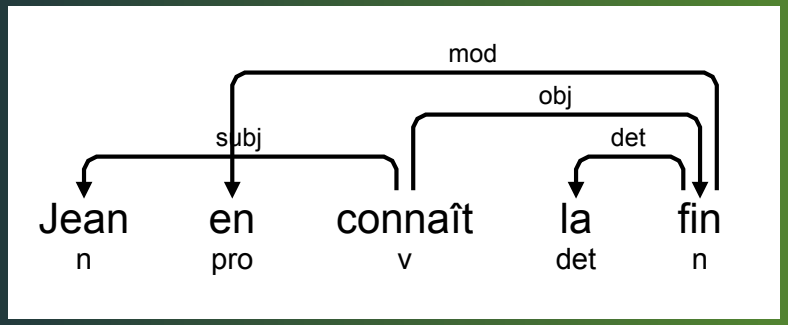
   a) Ecrire la définition d'une classe *Arbre* avec 3 attributs : *etiquette, fils_gauche, fils_droit*. Définir aussi une méthode *profondeur* qui permet de calculer la profondeur d'un arbre et une méthode *afficher* qui permet d'afficher un arbre

   b) Ecrire une classe ArbreSyntaxique qui dérive de la classe Arbre avec une méthode supplémentaire liste_mots qui permet d'extraire d'un arbre syntaxique la liste des mots constituant les feuilles.

   c) A partir des classes définies, écrire le programme qui permet d'afficher l'arbre syntaxique de la phrase « La belle brise la glace »

5.6 - Exercices

3. On veut écrire une classe pour les arbres de dépendance. Un arbre de dépendance est un arbre dont les nœuds sont étiquetés par les mots d'une phrase et dont les liens sont étiquetés par des fonctions grammaticales.Voici un exemple d'arbre de dépendance :



a) Ecrire la définition d'une classe *ArbreDep* avec 3 attributs : *mot, position, dependances*. L'attribut *mot* est une chaîne de caractères représentant le mot attaché à la racine de l'arbre. L'attribut *position* est un entier donnant la position dans la phrase du mot attaché à la racine. L'attribut *dependances* est une liste éventuellement vide de dépendances sous forme de couples *(étiquette, fils)*, où *étiquette* est une chaîne de caractères représentant le nom d'une dépendance et *fils* est l'arbre de dépendance cible de la dépendance. Définir aussi une méthode *champ* qui donne la liste des mots qui sont les feuilles de l'arbre de dépendance. qui permet de calculer la profondeur d'un arbre et une méthode *afficher* qui permet d'afficher un arbre

b) A l'aide de la classe ArbreDep, écrire le programme qui permet d'afficher l'arbre de dépendance de la phrase « Jean en connaît la fin »